Observations on Co-Array Fortran

Robert W. Numrich
Minnesota Supercomputing Institute
University of Minnesota, Minneapolis

Office of Science
U.S. DEPARTMENT OF ENERGY
University of Minnesota

Standards and Portability

- Co-arrays will be part of Fortran 2008
  - Shmem 1991
  - F-- 1991 (Fortran 77) Cray-T3D
  - SGI Origin 2000 (1996?) “-ufmm”
  - Co-array Fortran 1998   (Fortran 95)
  - Cray-T3E (1997?) Cray-X1   (-Z)
- Preprocessors and source-to-source translators don’t work
- Reference compiler doesn’t work
- Compiler implementation by more than one vendor is required.

The SPMD Model

- Gather/compute/scatter
- Bulk Synchronous Protocol
- MacroTasking
- MPI
- Shmem
- Co-Array Fortran
- UPC
- Titanium

Latency and Bandwidth

- Low latency machines are easier to program
- High bandwidth machines are harder to program
  - Cyber-205/Cray-1=20ns/12.5ns=1.6
  - Long vectors are hard to find
- Bandwidth
  - 8/t (byte/sec)
  - 1/νt (word/clocktick)
- Slow machines scale better
  - (flops/clocktick)/(words/clocktick)=flops/word

What makes programming hard?

- Science is hard
- Numerical methods are hard
- Boundary conditions are complicated and problem specific
- Geometries are complicated
- Data decomposition is hard, especially if it changes with time
- Mapping a logical decomposition to physical hardware is hard
- Memory management is hard
- Memory race conditions are easy to write, hard to find
- Message passing, either one-sided or two-sided, is hard
- Coordination/Synchronization is hard
- Load balancing is hard
- Computational scientists need to learn the tools of their trade.

Location, location, location

- The default for everything in CAF is local.
- No distributed data structures.
- Compiler optimizes local code.
- Deliberately designed to prevent the compiler form doing global optimization.
  - Dealing with local memory latency is the compiler’s job
  - Dealing with remote memory latency is the programmer’s job
- You need a map: local-to-global and global-to-local
  - Use the map only now and then
  - Otherwise everything is local

Global Address Space

- Hardware load/store instruction for any address in the machine.
  - When I need a word of data, I want to issue one instruction to get it
  - Compilers should be able to schedule loads
  - Stores should be free
- Minimal cache coherence
  - easiest coherence is to have no cache.
  - programmable local memory
- Don’t program it as a global address space.
  - If all addresses are potentially remote, the compiler has lost
  - Race conditions and memory consistency are nightmares!
- BlueGene with a global load/store would blow every other machine out of the water.

What is Co-Array Fortran?

- One simple extension to Fortran 95.
- All objects are local to a (virtual) image
- image: CAF's name for process, thread, task, rank, processor, core, cpu, pe, domain, locale or enclave.
- Some objects are marked with a co-dimension.
- Programmer can point from an object in one image to an object with the same name in another image through, and only through, the co-dimension.
- All communication is one-sided and explicit.
- All synchronization is explicit.

What is Co-Array Fortran?

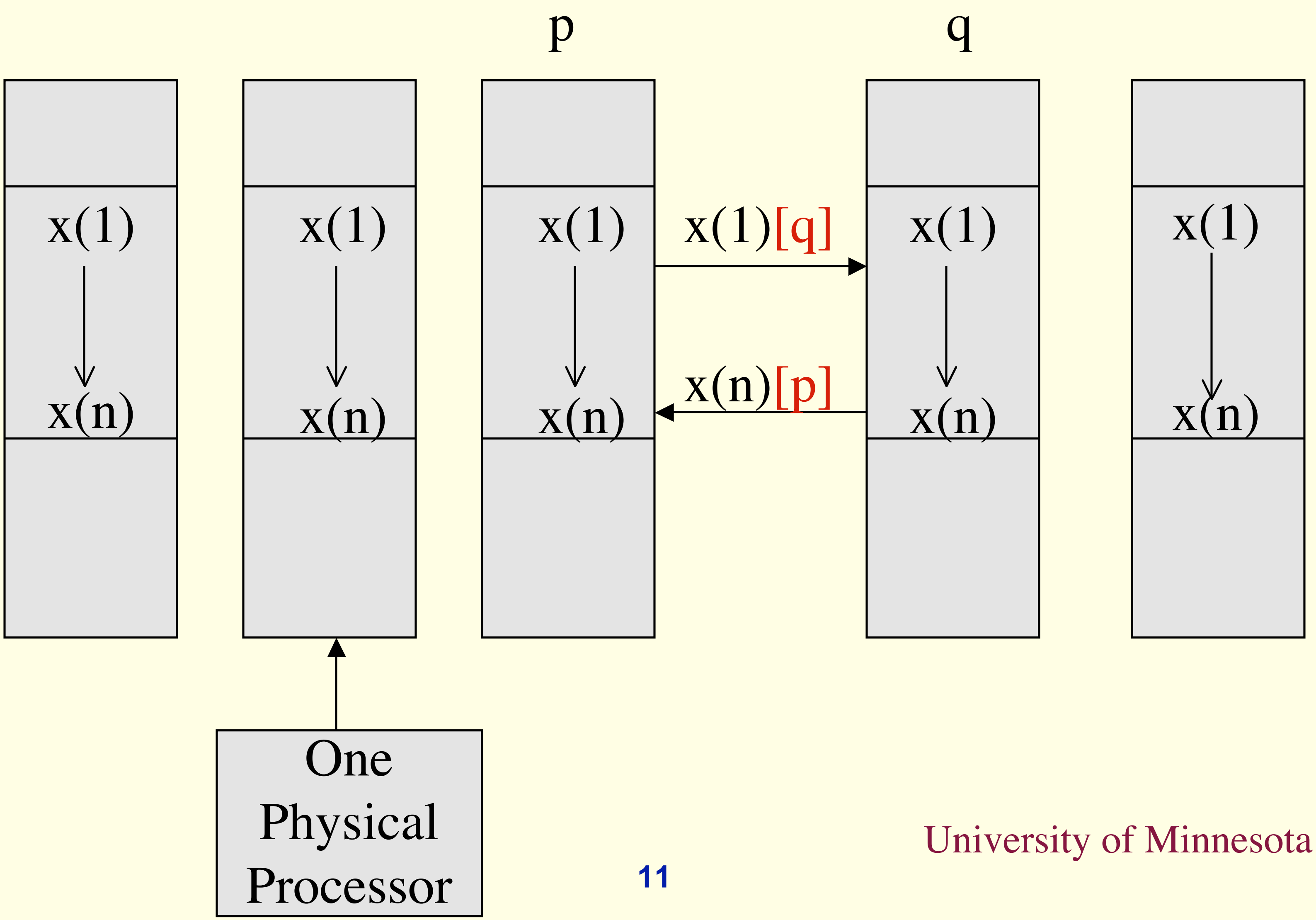
- Co-Array Fortran is a simple parallel extension to Fortran 90/95.
- It uses normal rounded brackets ( ) to point to data in local memory.
- It uses square brackets [ ] to point to data in remote memory.
- Syntactic and semantic rules apply separately but equally to ( ) and [ ].
- Co-Array syntax is a logical statement of my problem.
- The runtime system is responsible for mapping it onto hardware.

For Example

```
real :: y(n),x(n)[*]
y(:) = x(:)[p]
x(:)[q] = y(:)
x(:)[q] = x(:) + x(:)[p]
```

- Memory consistency?
- Blocking, non-blocking?

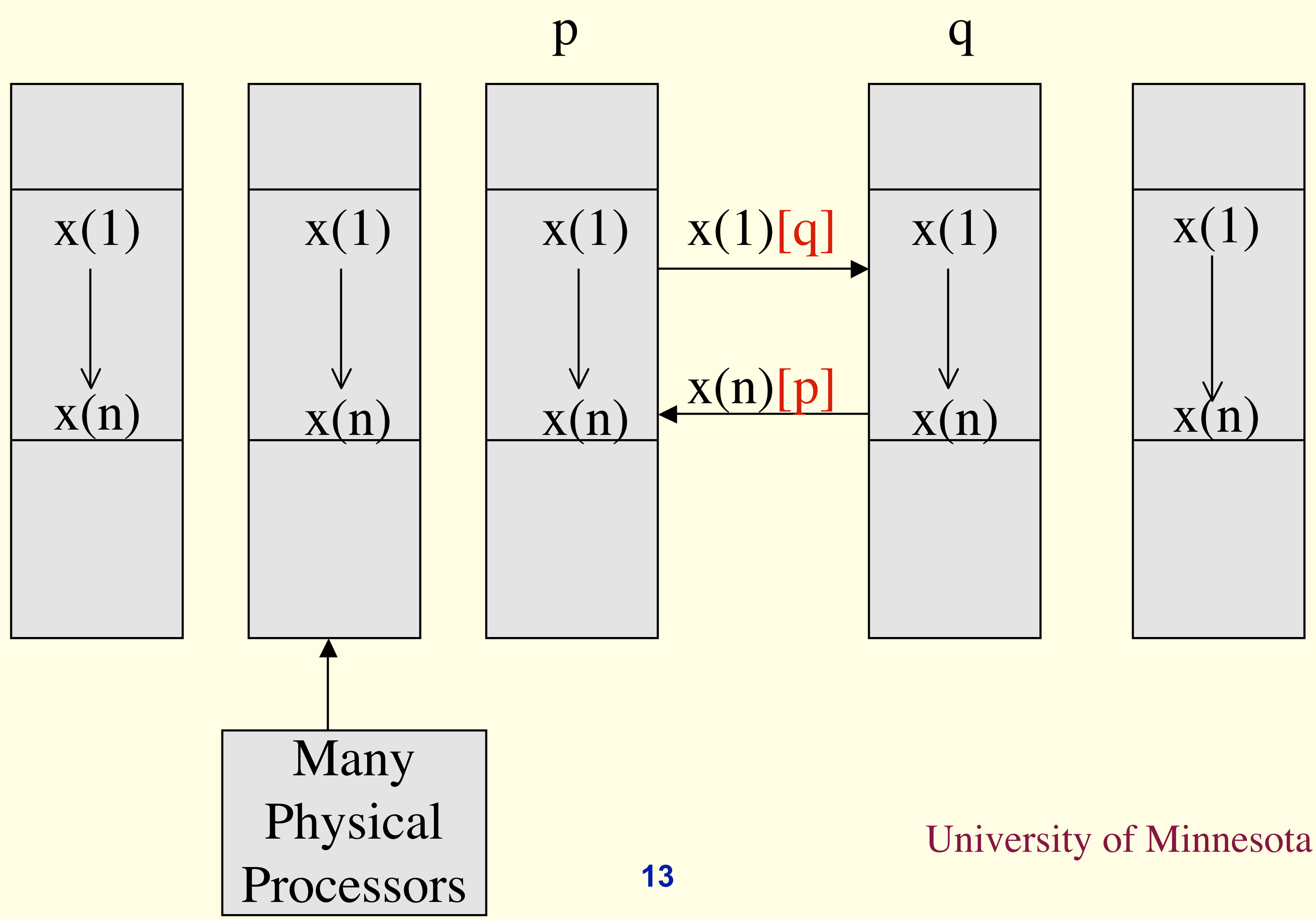
One-to-One Execution Model



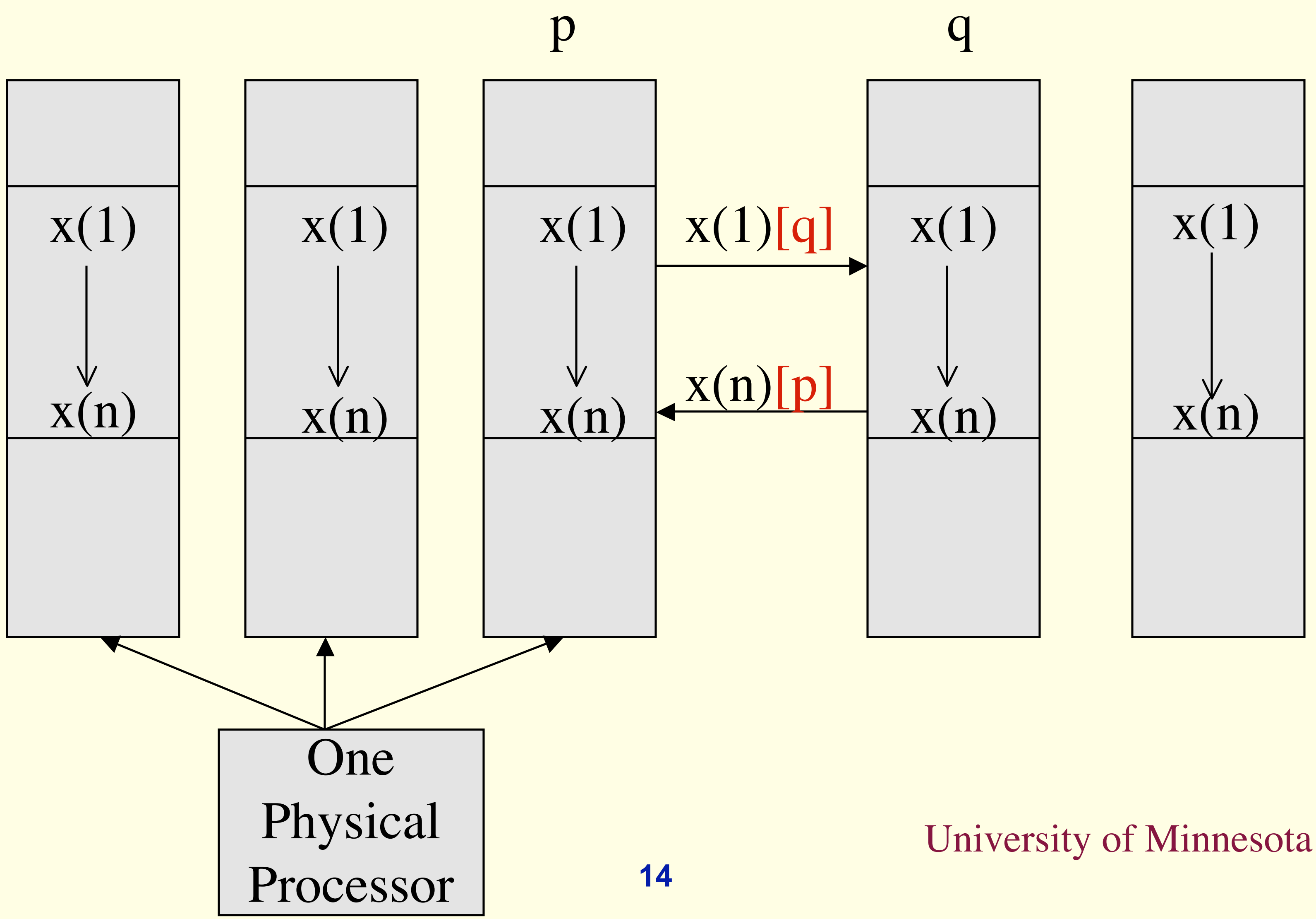
Load Balancing

- Over decomposition
  - Co-array data structures
  - Each image holds many patches
- Many more images than processors
- CAF does not specify how work is done
  - Work on local data
  - Go get remote data
  - Remote procedure invocation: x[p]%method()?

Many-to-One Execution Model

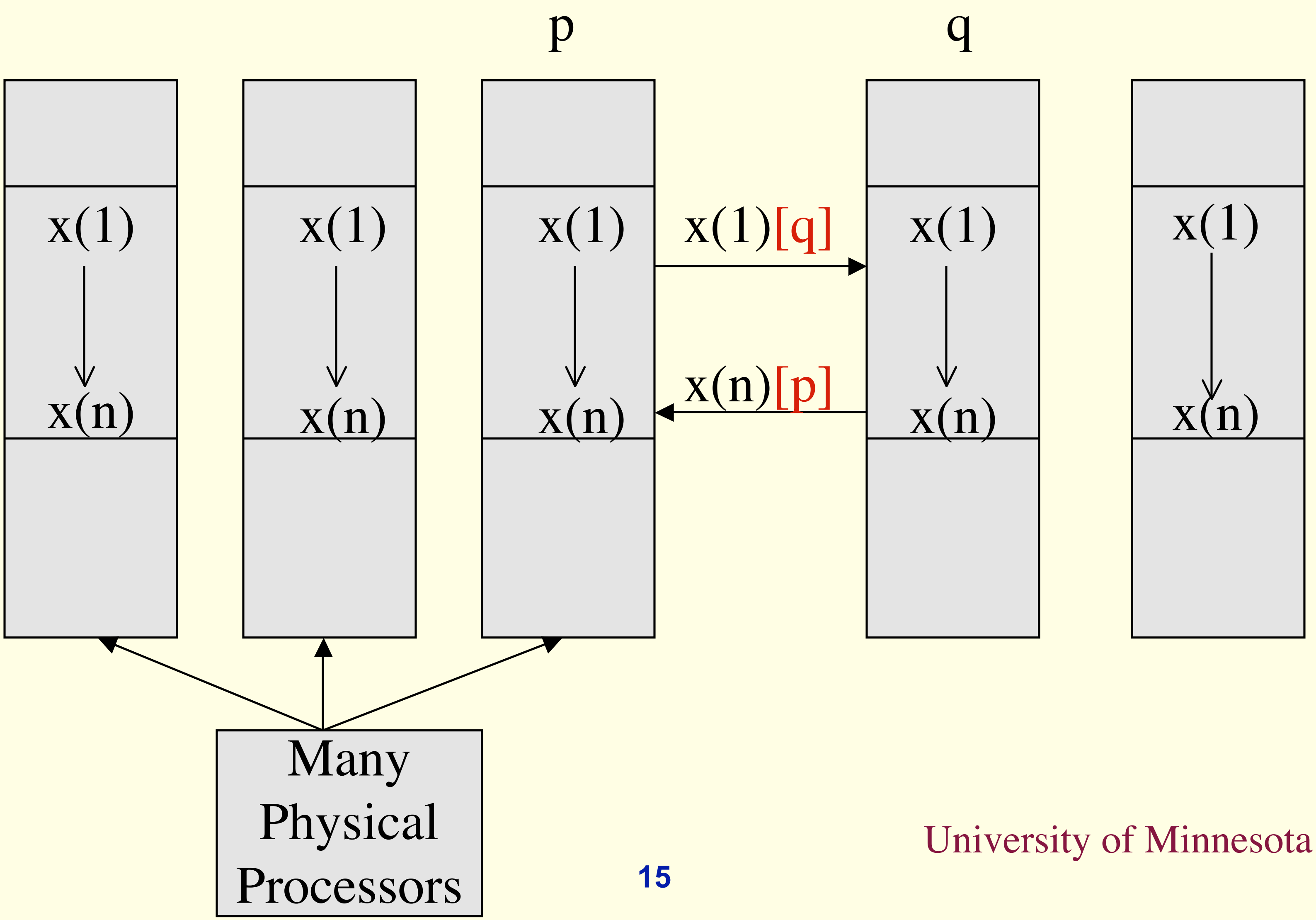


One-to-Many Execution Model



University of Minnesota

Many-to-Many Execution Model



Hierarchical Memory

real :: x[p,q,$*$]

y = x[:,r,s]   memory in first level

y= x[p,:,s]   memory in second level

y= x[p,r,:]   memory in third level

Why Did I Do It This Way?

- I can't write mirror-image code.
- It had to be easy to understand and natural for the Fortran language.
- It had to be simple to implement.
- It allowed the compiler to concentrate on local code optimization.
- I could write the code I wanted to write not code the compiler or a library wanted me to write.
- All communication is explicitly marked by the syntax.
- If what I wrote doesn't perform well, I can easily experiment with other ways of doing it.
- Memory race conditions happen only when I cause it to happen explicitly.

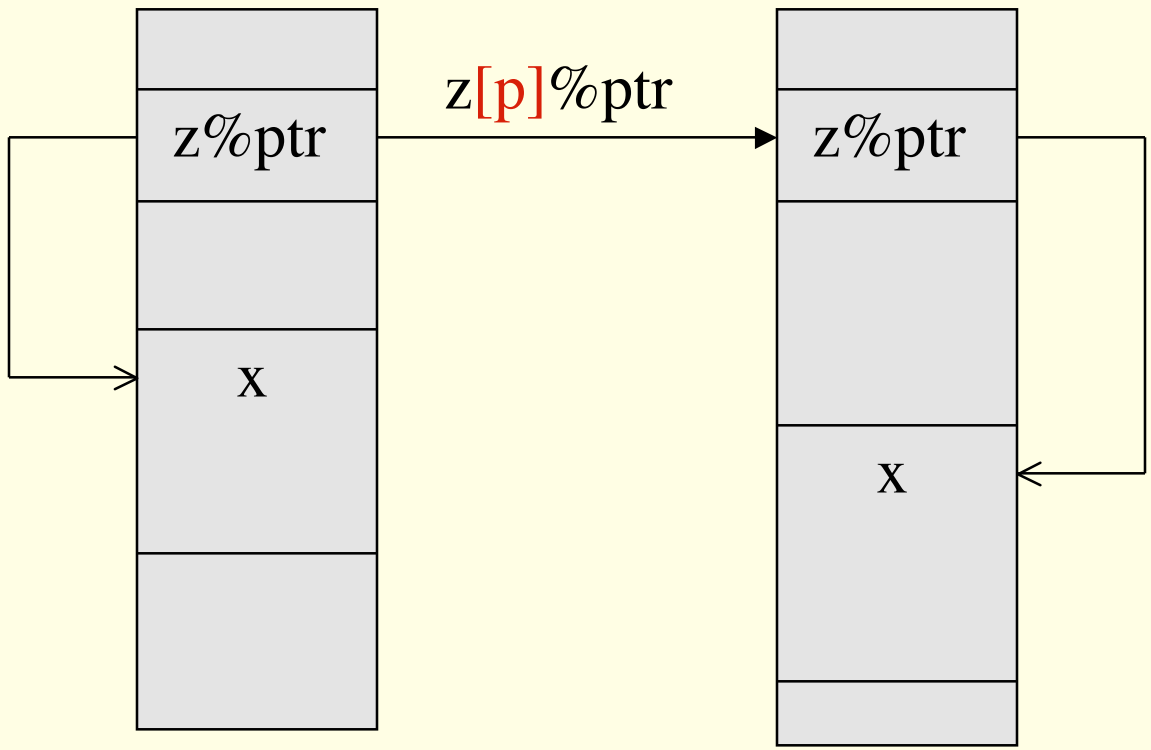
What I Don’t Like About It?

- Synchronization is hard to get right.
  - Machines with fast barriers and fast atomic operations no longer exist
  - Will transactions or full-empty bits really work efficiently on very large machines?
- Data decomposition has to be done manually.
- Work distribution has to be done manually.

Using “Object-Oriented” Techniques with Co-Array Fortran

- Fortran 95 is not an object-oriented language.
- But it contains some features that can be used to emulate object-oriented programming methods.
  - Named derived types are similar to classes without methods.
  - Modules can be used to associate methods loosely with objects.
  - Generic interfaces can be used to overload procedures based on the named types of the actual arguments.

Irregular and Changing Data Structures



For Example …

```
type(BlockMatrix) :: a[*]
type(MatrixMap)   :: map
call newMatrixMap(n,m,k,l,p,q)
call newBlockMatrix(a,map)
call luDecomp(a)
call writeBlockMatrix(a)
call deleteMatrixMap(map)
call deleteBlockMatrix(a)
```

LU Decomposition

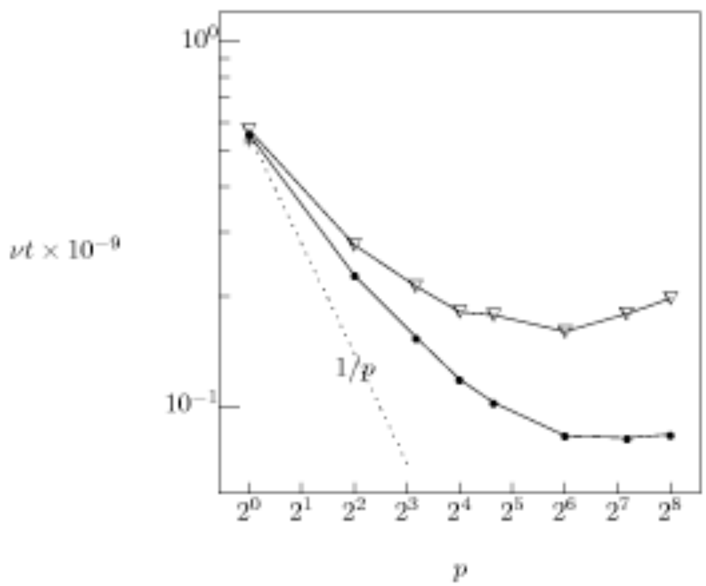


Figure 6: Time as a function of the number of processors $p = q \times r$ for block-cyclic LU decomposition. The matrix size is 1000×1000 with blocks of size 48×48. Time is expressed in dimensionless giga-clock-ticks, $\nu t \times 10^{-9}$, as measured on a CRAY-T3E with frequency $\nu = 300$MHz. The dotted line represents perfect scaling. The curve marked with bullets ($\bullet$) is code written in Co-Array Fortran. The curve marked with triangles ($\triangledown$) is SCALAPACK code.

Why Language Extensions?

- The language itself need not contain high levels of abstraction.
- The programmer defines objects that fit the problem.
- Runtime system maps them onto hardware.
- Compiler evolves as the hardware evolves.
  - Lowest latency allowed by the hardware.
  - Highest bandwidth allowed by the hardware.
  - Data ends up in registers or cache not in memory
  - Arbitrary communication patterns
  - Communication along multiple channels

Simple Things That Would Improve My Productivity

- Fast compile time
- Interactive prototyping mode
- Interactive access to the machine
- Trace back when the program aborts
- Instruction level real-time clock
- Print with node number attached
- Easy graphical display
- Single-processor performance provided by the compiler